VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Implementation of Parser for RDF Data Files

Implementace parseru pro soubory s RDF daty

2016

Tomáš Michalek

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student:	**Tomáš Michalek**
Study Programme:	B2647 Information and Communication Technology
Study Branch:	2612R025 Computer Science and Technology
Title:	Implementation of Parser for RDF Data Files Implementace parseru pro soubory s RDF daty
The thesis language:	English

Description:

RDF is W3C specification originally created to model metadata. Each RDF triple has following structure: subject-predicate-object. Aim of this thesis is to implement parser, able to read various formats used to store RDF data. Parser is supposed to read commonly used formats and print out warning in case of errors. Output of the parser will be sequence of extracted triples.

1. Research formats used to store RDF data.
2. Study current existing solutions to parse such files.
3. Design and implement parser for those formats.
4. Measure performance of implementation and compare it to existing solutions.

References:

[1] BECKETT, Dave; MCBRIDE, Brian. RDF/XML syntax specification (revised). W3C recommendation, 2004, 10
[2] BECKETT, David; BERNERS-LEE, Tim. Turtle-terse RDF triple language. W3C Team Submission, 2008, 14
[3] BECKETT, Dave; BARSTOW, Art. N-Triples. W3C RDF Core WG Internal Working Draft, 2001

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Roman Meca**

Date of issue: 01.09.2014
Date of submission: 29.04.2016

[illegible]

doc. Dr. Ing. Eduard Sojka
Head of Department

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA OSTRAVA
Fakulta elektrotechniky a informatiky
1D16

[illegible]

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

[illegible]

Ostrava, 27 April 2016 ....

I would like to thank my supervisor ing. Roman Meca for his help with completing this thesis. I would also like to to thank my family for understanding and support while I was writing it.

Abstrakt

Tato práce se zabýva implementaci RDF parseru pro formáty Turtle a N-Triples. Výsledkem práce je implementace push (event) parseru, kde lexer a parser jsou vytvořeny dvěmi nástroji: Ragel a Lemon. Zároveň se věnuje podpoře UTF-8 a využívá možnosti clang compileru.

Klíčová slova: RDF, Turtle, N-Triples, parser, lexer, Lemon, Ragel, manipulace data

Abstract

This thesis focuses on implementation of parser for RDF data in Turtle and N-Triples formats. The resulting implementation is push (event) parser with lexer and parser created by two separate tools: Ragel and Lemon respectively. It also deals with correct support of UTF-8 encoding as well as uses Clang compiler features.

Key Words: RDF, Turtle, N-Triples, Parser, Lexer, Lemon, Ragel, data manipulation

Contents

List of symbols and abbreviations

ABI	–	Application Binary Interface
API	–	Application Programming Interface
BMP	–	Basic Multilingual Plane
CFG	–	Context-Free grammar
CFL	–	Context-Free language
CI	–	Continuos Integration
CLT	–	Command Line (Developer) Toolkit
DFA	–	Deterministic Finite Automata
FA	–	Finite Automata
FPU	–	Floating-point Unit
GCD	–	Grand Central Dispatch
JSON	–	JavaScript Object Notation
N3	–	Notation 3
OS	–	Operating System
PIII	–	Pentium III
RDF	–	Resource Description Framework
SIMD	–	Single Instruction Multiple Data
SSE	–	Streaming SIMD Extensions
URI	–	Universal/Uniform Resource Identifier
VM	–	Virtual Machine
W3C	–	World Wide Web Consortium
XML	–	Extensible Markup Language

List of Figures

List of Tables

Listings

1 Introduction

Where the human can easily extrapolate informations from the webpages, machines have much harder time doing the same. This forced the applications to be optimized for particular set of data and attributes.

In order to deal with this problem, new extension for the web was designed called **semantic web**, which allows to extract the data by humans, furthermore it allows the machines to do the same easily and effectively. One of the building blocks of *semantic web* is **Resource Description Framework** (RDF), which is used for the description of the relationships between of the entities and data sources. RDF describes those relationships using so called triples, which are then expanded into graphs.

The XML became the base format for describing the RDF graphs, creating RDF specific flavour RDF/XML. Thanks to the markup nature of the format, it is easily processed by variety of tools, as XML parsing is well known subjects. The downside of the XML format is the lowered readability for the humans as well as relatively bigger file-sizes because of additional markup of data. To address those issues, number of other formats has been created. Those formats use shorthands and more compact notations to express the same data as their RDF/XML counterpart.

The aim of this thesis was to implement parser, which would be able to read alternative notations and print out the constructed triples to user. At the same time it should be able to detect corrupted input and notify user about the occurred data inconsistency.

As the focus of implementation is Turtle format, it's secondary focus is Turtle's subset N-Triples. At the same time the possibility of expansion to other overlapping formats such as N-Quads and TRiG was kept in mind while designing the parser to allow future expansions.

The text of this thesis is divided into the following chapters:

- **Chapter 2** gives introduction to theoretical background required to parser RDF formats.
- **Chapter 3** touches upon the already existing tools used to parse the RDF data. Select the tools which are then used as the base for performance analysis.
- **Chapter 4** describes practical choices made while developing the implementation parser, dependencies and auxiliary support tools.
- **Chapter 5** summarises the status of the implementation, its performance and optimization efforts.
- **Chapter 6** summarises the thesis as a whole as well as presents critique of the several points, where the current implementation runs short. It also presents ideas for future work.

Complete version of the implementation code described in this thesis, can be found at the following website: `https://www.github.com/michto01/ardp` as well as enclosed CD.

2 Preliminaries

This chapter deals with the necessary terminology, concepts and backgrounds used throughout this thesis. It shows general ideas and approaches to general data processing via lexing and parsing and RDF data formats for storing the triples.

2.1 Formal Language

In computer science and mathematics we denote *formal language* as infinite set of strings contained by specific rules assigned to them. The language then specifies various data structures used in/by the machine. The formal definitions allow the machine, which know the rules to extract the data and transform them to another format.

Definition 1 *A **string** s over an alphabet Σ is a finite sequence of symbols drawn from this alphabet. The **length** of a string s is the number of symbols is s. The **empty string**, denoted as ϵ, is a special string of length zero.*

Definition 2 *A **language** L over alphabet Σ is a set of strings over this alphabet.*

Regular expressions

An algebraic-expression notation for describing the regular languages[1], which can be recognised by Finite Automata. The following definitions specify the **regular expressions**. Regular expressions provide a way of specifying simple languages whose structure may involve sequencing, choice, and repetition, but no recursive syntactic constructs.

Definition 3 *R is regular expression over alphabet Σ exactly if R is one of the following:*

1. *$\emptyset$ is a regular expression denoting the language $\{\emptyset\}$, the empty set.*
2. *ϵ is a regular expression denoting the language $\{\epsilon\}$, the set containing only empty string - String without characters and length 0.*
3. *If c is a symbol in Σ, then c is regular expression denoting the language $\{c\}$, the set containing the literal character c.*

Definition 4 *Given regular expressions R and S, following operations over them are additionally defined to produce the expressions:*

1. ***concatenation** RS denotes language $L(r)^\circ L(s)$, the set obtained by concatenating the strings in R with strings in S.*
   (Example 1)

[1]Mark E. Gold has shown, that not every regular language can be described in such way, but for purposes of this thesis, such cases are ignored as they do not occur in any of the used RDF specifications.

2. ***alternation*** *$R|S$ denotes $L(r) \cup L(s)$, the set union of sets defined by R and S.*
   (Example 2)

3. ***Kleene closure*** *R^* denotes language $(L(r))^*$, the smallest superset of set defined by R that contains ϵ and is closed under string concatenation.*
   (Example 3)

In order to simplify the notation of regular expression, few additional rules are added. To avoid need for unnecessary parentheses, the precedence is given to the operators. The Kleene closure $*$ operator has the the highest precedence, followed by the concatenation then union.

Two operators, '+', where $r+ = rr^*$, and '?', where $r? = r|\epsilon$ *(Examples 4 and 5)*, are added to aid readability of notation. Both of them have the same precedence as *Kleene* closure. Following examples showcase aforementioned expression and the languages they denote:

Example 1
$\{"ab", "c"\}\{"d", "ef"\} = \{"abd", "abef", "cd", "cef"\}$ ■

Example 2
$\{"ab", "c"\}|\{"ab", "d", "ef"\} = \{"ab", "c", "d", "ef"\}$ ■

Example 3
$a^* = \{\epsilon, a, aa, aaa, ...\}$ ■

Example 4
$a^+ \equiv aa^* = \{a, aa, aaa, ...\}$
$01 + 0 = \{010, 0110, 01110, ...\}$ ■

Example 5
$a?b \equiv ab|b = \{ab, b\}$
$colou?r = \{color, colour\}$ ■

Although recognizing strings defined by regular expressions is not nearly as straight-forward as generating strings from them, the recognition problem is not that difficult and can always be done in linear time, proportional to the length of the input string. The relevant algorithms can be found in any compiler textbook [16].

Because of their conciseness, simplicity, and efficiency, regular expressions have become the *de facto* basis for generic searching and text manipulation tools of all kinds, as well as for defining the lexical stage of many conventional language processors.

Issues of ambiguity and information preservation are not usually ignored for regular expression processing, because in practice, regular expressions are primarily used merely to find or identify strings matching a pattern for subsequent processing by other methods, and not to extract information from those strings directly.

Deterministic Finite Automata

Deterministic Finite Automata (DFA) is finite state machine (FSM) that accepts or rejects finite strings of symbols and only produces a unique path through the machine for each particular input string. 'Deterministic' refers exactly to such uniqueness.

As DFAs recognize exactly the set of regular languages, it is used in lexical analysis and pattern matching.

Definition 5 *A **Deterministic Finite Automata** is a mathematical model represented by 5-uple $(Q, \Sigma, \delta, q_0, F)$, where:*

1. *Q is a finite set of states.*
2. *Σ is a finite set of input symbols called alphabet.*
3. *δ is a transition function $\delta : Q \times \Sigma \Rightarrow Q$, that maps state-symbol pairs to a state.*
4. *q_0 is a initial state $q_0 \in Q$.*
5. *F is a set of final (accepted) states $F \subseteq Q$.*

DFA recognises an input string $q = c_1c_2...c_n$ iff final state q_n exists, where $q_{i+1} = \delta(q_i, c_{i+1})$, starting with with the initial state q_0.

The informal definition states that DFA is an automaton, which defines *at most* one transition for each state and each input symbol; the transition function is allowed to be partial. When no transition is defined, such automaton halts.

Regular expressions and DFAs are proven to be equivalent[2], in the sense that for every regular expression there exists a DFA that is able to recognise the same language, and vice versa.

In practice, to improve the readability of the source by humans, the regular expressions are used to provide rules, which various tools transform into DFA to improve performance, as DFAs are easy and cost-efficient to implement.

2.2 Grammars

The structure of the input languages is described with a *context-free grammar* (CFG) or grammar for short as only *context-free languages* (CFL) are discussed in this thesis.

Definition 6 *A **grammar** is quadruple (Σ, V, S, P), where:*

1. *Σ is a finite nonempty set called the **terminal alphabet**. The elements of Σ are called the **terminals**.*

[2] `http://grid.cs.gsu.edu/~cscskp/Automata/FA/node19.html`

2. *V is a finite nonempty set disjoint from Σ. The elements of V are called the* ***nonterminals*** *or* ***variables****.*

3. *$S \in V$ is a distinguished nonterminal called the* ***start symbol****.*

4. *P is a finite set of* ***productions*** *(or* ***rules****) of the form $\alpha \Rightarrow \beta$ where $\alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$, i.e. α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.*

Definition 7 *Let (Σ, V, S, P) be a grammar. A sentential form of G is any string of terminals and nonterminals, i.e. a string over $(\Sigma \cup V)$.*

Definition 8 *Let (Σ, V, S, P) be a grammar, and let γ_1, γ_2 be two sentential forms of G. We say that γ_1* ***directly derives*** *γ_2, written $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \Rightarrow \beta$ is a production in P.*

Definition 9 *Let γ_1 and γ_2 be two sentential forms of a grammar G. We say that γ_1* ***derives*** *γ_2, written $\gamma_1 \Rightarrow *\gamma_2$, if there exists a sequence of (zero or more) sentential forms $\sigma_1, \cdots, \sigma_n$ such that*

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2.$$

The sequence $\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$ is called a ***derivation*** *of γ_2 from γ_1.*

Definition 10 *Let (Σ, V, S, P) be a grammar. The language generated by G, denoted by $L(G)$, is defined as $L(G) = \{x | x \in \Sigma^*, S \Rightarrow^* x\}$.*

The words in $L(G)$ are also called the sentences of $L(G)$.

2.2.1 Hierarchy of Grammars

Standard division of grammars, defined by Chomsky, also called *Chomsky hierarchy*, divide grammars into four classes. Each class in succession increases the restrictions on the form of the productions.

Definition 11 *Let $G = (\Sigma, V, S, P)$ be a grammar.*

1. *G is also called a* ***Type 0*** *grammar or an* ***unrestricted*** *grammar.*

2. *G is a* ***Type 1*** *or* ***context-sensitive*** *grammar is each production $\alpha \Rightarrow \beta$ in P satisfies $|\alpha| \leq |\beta|$. By "special dispensation," we also allow a Type 1 grammar to have the production $S \Rightarrow \epsilon$, provided S does not appear on the right-hand side of any production.*

3. *G is a* ***Type 2*** *or* ***context-free*** *grammar if each production $\alpha \Rightarrow \beta$ in P satisfied $|\alpha| = 1$; i.e., α is a single nonterminal.*

$$\begin{array}{lcl} E & \Rightarrow & N \\ & | & \text{'('}\ E\ \text{'+'}\ E\ \text{')'} \\ & | & \text{'('}\ E\ \text{'}-\text{'}\ E\ \text{')'} \\ N & \Rightarrow & D \\ & | & DN \\ D & \Rightarrow & \text{`0`}|...|\text{`9`} \end{array}$$

Figure 1: Context-free grammar for trivial arithmetic expression language

4. *G is a* ***Type 3*** *or* ***right-linear*** *or* ***regular*** *grammar if each production has one of the following three forms:*

$$A \Rightarrow cB, A \Rightarrow c, A \Rightarrow \epsilon,$$

where, A, B are nonterminals (with $B = A$ allowed) and c is a terminal.

The language generated by a Type-i, where $i = 0, 1, 2, 3$, are called Type-i languages. A Type 1 language is all called **context-sensitive language** (CLS), and a Type 2 language is also called a **context-free language** (CFL).

The four classes of languages in the Chomsky hierarchy have also been completely characterized in terms of Turing machines and natural restrictions on them, which helps to predict, if the particular language will be parsed efficiently.

2.2.2 Context-free Grammar

The regular expressions, mentioned previously, have important limitation in their expressiveness. Namely inability to express recursive syntactic constructs. The syntax of most languages inherently contains such recursive structures as, *"If e_1 and e_2 are expression, then '$e_1 + e_2$' is also an expression"*. Even the informal definition of regular expression uses such rules, and as such, regular expression are not powerful enough to express their own syntax effectively.

The *context-free grammar* (CFG) brings solution to this problem by representing the syntax by set of rules that can refer to each other recursively as opposed to one rule. CFG utilizes two classes of symbols: **terminals**, representing the atomic syntactic elements, which correspond to literal characters in regular expressions; and **nonterminals**, representing higher-level, composite structures such as expressions and statements.

As regular expressions, the CFG in its essence provides recipe for expressing the string in language, defined by that grammar. For example *Figure 1* shows trivial CFG for trivial expression language, which contains addition, subtraction, and decimal numbers.

The grammar used three nonterminal symbols: E representing the expressions, N representing the decimal number represented by one or more decimal digits represented by D. If one wants to generate an expression (E), the grammar gives three options:

1. Write the number using rule N.

2. Write string in form '$(e_1 + e_2)$'. e_1, e_2 are generated by E

3. Write string in form '$(e_1 - e_2)$'. e_1, e_2 are generated by E

Similarly, to generate a number (N), there are two choices: write a single digit or prepend a digit (D) to number already generated by rule this rule (N).

Opposite to regular expressions, ambiguity is a much more important problem in practice for CFGs, because the primary goal is usually not just to recognize whether a string conforms to a particular CFG, but to parse the string, effectively reconstructing the full set of decisions (that would have been) used to generate the string from the CFG.

Both recognizing and parsing CFGs is a much harder problem than recognizing regular expressions. Parsing arbitrary CFGs is equivalent in complexity to boolean matrix multiplication, which is generally believed to lie between $O(n^2)$ and $O(n^3)$. The fastest known algorithm to recognize arbitrary unambiguous CFGs is $O(n^2)$, but the question of whether an arbitrary CFG is unambiguous is undecidable in general [17].

2.2.3 Backus-Naur Form

In the most common practical representation for context-free grammars, known as *Backus-Naur Form* (BNF), each rule has the form $n \Rightarrow v_1|...|v_n$, where n is the nonterminal symbol that the rule defines, and each v_i on the right-hand side is a string consisting of any number of terminals and/or nonterminals. It was first used to define ALGOL 60.

Each v_i in the rule represents an alternative syntactic expansion or derivation for the nonterminal on the left. No significance is attached to the order in which multiple alternatives appear in a rule, or to the order in which rules appear in the grammar. Here is example of BNF grammar notation:

```
<name>     ::= tom | mike | judy
<sentence> ::= <name> | <list>
```

Listing 1: Example of BNF grammar notation.

This form's main properties are the use of angle brackets to enclose non-terminals and of ::= for "may produce". In some variants, the rules are terminated by a semicolon.

This thesis uses the extended BNF (EBNF) which adds the regular expression operators: (), *, ?, $^+$, as it is primary format used by W3C, which in turn is responsible for all of the grammar specifications used in this thesis.

2.3 Parser

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree[14]. The parser usually has 3 distinct stages:

- Lexical analysis
- Syntactic analysis
- Semantic parsing

2.3.1 Lexical Analysis

Lexical analysis is process in which input string stream of characters is broken down into groups of meaningful characters called *lexemes*. It is the first phase of parsing. Other names conveying the same intent are "Lexer", "Scanner" or "Tokenizer".

For each lexeme, lexer outputs a *token*. Token is pair (*token name*, *value*). The *token name* is usually abstract symbol used in syntax analysis. The *token value* contains actual content from file. Tokens are then passed to the next stage of parser: syntax analyser.

The lexer uses grammar to specify the tokens terminals and is usually implemented in form DFA or/and Regular expressions.

2.3.2 Syntactic Analysis

Syntactic analysis tests if the acquired tokens form meaningful data. To asses this syntax analyzer uses *context-free grammar* that defines algorithmic procedures for components. The components form an expression, in which the components follow specific order to satisfy the rule are stated.

2.3.3 Semantic Analysis

Semantic analysis is the last part of the process of parsing. It uses the syntax tree and the information in the symbol table to check the input against the language definition.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

2.3.4 Parser types and strategies

Since the 60's many scientists studied parsers and developed techniques to speed up parsing, define it's boundaries and optimize the steps of the parsing process. The studies shown that the parsing algorithms fall into two basic categories **top-down** and **bottom-up** parsers.

As indicated by their names, a *top-down parser* builds derivation trees from the top (root) to the bottom (leaves), while a *bottom-up parser* starts from the leaves and works up to the root. Although neither method is good for handling all *context-free grammars*, each provides efficient parsing for many important subclasses of the *context-free grammars*, including those used in most programming languages. In this thesis only the unambiguous grammars are discussed.

Typical parser has an input buffer, a pushdown stack, a parsing table, and an output stream. The input buffer contains the string to be parsed. The stack contains a sequence of terminals or nonterminals.

2.3.4.1 Top-down parsing is the strategy, where the parser starts from the top of the highest level of the parser tree and works its way down the parser tree by employing the rewrite rule of the specified grammar. Tokens are consumed from left to right using so called leftmost derivation.

Some grammars may have entries containing multiple productions. It would be possible for parsing algorithm to make a wrong choice and incorrectly report a string as not being derivable, and backtracking to the last choice to try another would increase the run time, rendering it inefficient. For such purposes the lookahead mechanism was created. *Lookahead* uses k next tokens to determine current token production.

A context-free grammar whose parsing table without multiple defined entries called **LL(1) grammar**. The numbers signifies the fact that the LL parser uses one input symbol lookahead to decide its next move. For any constant $k > 1$, a grammar is said to be an **LL(k) grammar** if its parsing table using k lookahead symbols has no multiply defined entries.

Although LL(k) grammars form a larger class than LL(1) grammars, there are still grammars that are not LL(k) for any constant k. The texts [17][16] provide several techniques for dealing with non-LL(k) grammars, such as grammar transformations and backtracking. When backtracking is used, the parsing process is often called *recursive-descent parsing*, and can be very time consuming due to the use of many recursive calls.

2.3.4.2 Bottom-up parsing , is the strategy, where the parser starts from the bottom left end of the parsing tree and incrementally works its way up and rightwards by composing the larger structures from the nodes until it reaches the root node. The strategy waits until all the parts of construct are in place before committing to what the construct actually is. As this strategy commonly do not require backtracking, it is more used in computer language grammars then top-down parsing.

Basic implementation is LR parser. The **LR** stands for Left-to-Right input scanning, the rightmost derivation. For any input string x, the LR parser scans x from left to right and tries to find the reverse of the sequence of productions used in the rightmost derivation of x.

In practice, the rightmost derivation is easier to process then leftmost derivation. LR parsing according to [17] is especially preferred because:

1. in can handle virtually all programming language constructs;
2. it has very efficient implementations;
3. it is more powerful than the LL parsing;
4. it detect syntactic errors quickly;

The principal drawback is that the LR parser construction is very involved. Fortunately, there exists efficient algorithm that can automatically create *context-free grammar* based LR parsers. As whole description of the process goes outside of scope for this thesis, one should refer to [17] for construction of such parser.

2.4 Resource Description Framework

Resource Description Framework (*RDF*) is *W3C* specification for description and modeling of objects, their relationships and properties, created in 1999. At first the RDF was intended to represent metadata, but it's now used as universal tool for modeling information or sources in many different syntaxes. Data stored in RDF format are formed by 'TRIPLETs: SUBJECT - PREDICAMENT - OBJECT'. The triplet tells us:

"SUBJECT has property PREDICAMENT and the predicament has value of OBJECT"

The triples are populated into oriented graphs. One of the interesting properties of RDF is ability to deduce additional information - that is, creation of triplets not (physically) existing in the RDF storage[2][3].

2.4.1 Relationship with the Semantic Web

Semantic web is all about the data. On the internet there are large quantities of the data available in different forms. The most dominant forms are still texts, images and tables. Those

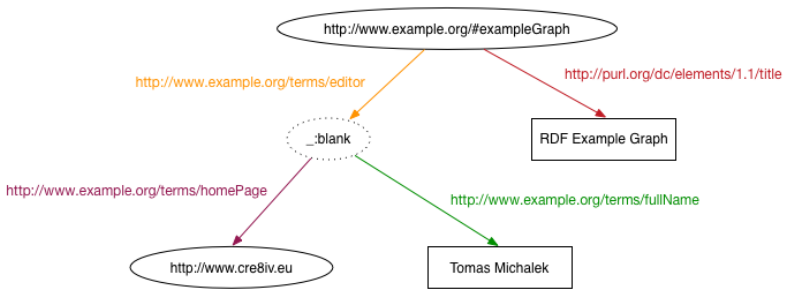
http://www.example.org/#exampleGraph
http://www.example.org/terms/editor
http://purl.org/dc/elements/1.1/title
_:blank
RDF Example Graph
http://www.example.org/terms/homePage
http://www.example.org/terms/fullName
http://www.cre8iv.eu
Tomas Michalek

Figure 2: Example RDF graph which describes website and its author.

are easy to understand for human, as one can draw his own logical conclusions about the data and links them together.

Machines do not possess such ability to do logical deductions, so how it can then recognize relationships between information? This exact problem is challenged by ‘Semantic Web‘. The basic principle is to complement the human readable form (text, image ...) by metadata, from which the machine will be able determine relationships between objects.

The semantic web development is driven by the World Wide Web Consortium and it's technologies specifications contains specification for *RDF* and *OWL* etc.

2.4.2 Graph representation

As was previously stated, the data contained in *RDF metadata* can be look upon as on oriented graph. Example of such graph is shown in *Figure 2*. The root node is represented by subject, edge is predicament and the result node is object. The nodes can be of following types:

- Entities (has its own definition in dictionary, has its own URI)
- Literals
- Blank nodes (not identified by URI)

2.4.3 RDF terms

2.4.3.1 URI stands for the **U**niversal **R**esource **I**dicator and its used to unequivocally identify a resource. The two subset used in this work are URL and URN. **URL** indicates access to the resource, whereas **URN** indicates resource name.

URI according to RFC 3986 is defined as:

scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]

https://www.example.org/

2.4.3.2 BLANK NODE

is RDF graph node representing arbitrary resource for which literal or URI are not given. The resource represented in such manner is also called anonymous resource. According to RDF specification such node cannot be predicate. Common use is to represent complex attributes without the need to name explicitly the auxiliary nodes.

2.4.3.3 LITERAL

is RDF graph node representing value. It can only be used as triples object. The RDF Schema recognises several basic types of literals such as string, boolean, integer, decimal, float. Other types are represented as a string with explicitly stated datatype using HAT (^^) notation.

2.4.4 Containers

RDF has ability to work with *containers*, which can hold sets of elements.

- *rdf:Alt* - set of alternative values; Used for example to describe alternative languages, to which the book is translated.
- *rdf:Bag* - unordered set of values; Can contain the same value multiple times.
- *rdf:Seq* - ordered set of values; Can contains some values multiple times.

Container is represented using the blank node. The blank node then has predicate with its type (*rdf:type*). The items of such container are marked with predicate *rdf:li* or for particular elements, predicates *rdf:_1*,*rdf:_2*, $\cdots$, *rdf:_n*.

2.4.5 Collections

One of the shortcomings of the containers is the inability to detect is the containers have some other elements. For this purpose, RDF collection (also called lists) were created. They are used to store all the values belonging to particular set.

The collections are implemented in RDF as linked lists. Every element in collection has item pointed to by predicate *rdf:first* and then predicate *rdf:rest*, which points to next element in collection. The last item of the collection points to *rdf:nil*. Example graph of collection is shown is *Figure 3*.

2.4.6 Serialization formats

For exchanging the *RDF metadata* there are numerous file formats in use. The main focus of this thesis in on the subsets of `Notation 3`: *Turtle* and *N-Triples*. Other well known formats are: `RDF/XML`, `RDF/JSON`, `TriG` and `N-Quads`.

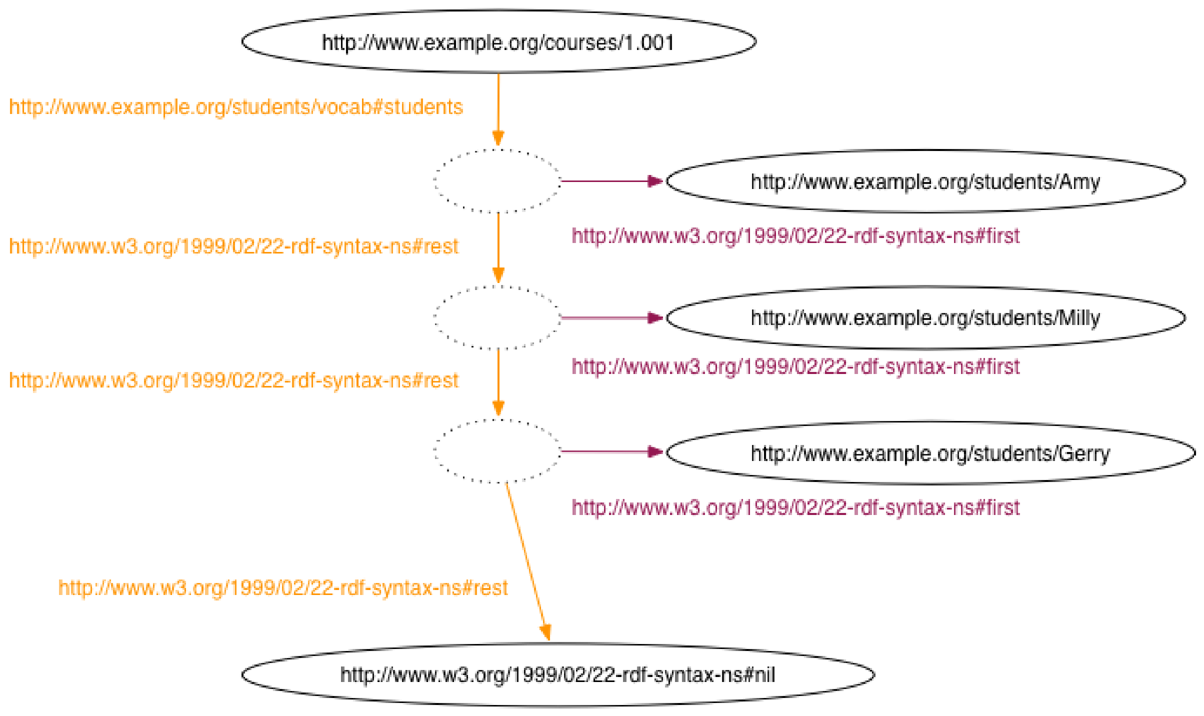
http://www.example.org/courses/1.001
http://www.example.org/students/vocab#students
http://www.example.org/students/Amy
http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
http://www.w3.org/1999/02/22-rdf-syntax-ns#first
http://www.example.org/students/Milly
http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
http://www.w3.org/1999/02/22-rdf-syntax-ns#first
http://www.example.org/students/Gerry
http://www.w3.org/1999/02/22-rdf-syntax-ns#first
http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
http://www.w3.org/1999/02/22-rdf-syntax-ns#nil

Figure 3: Graph representing example RDF collection

2.4.6.1 RDF/XML XML is one of the most used formats in data exchange in computers today. It's specification is made and maintained by the W3C, so specification of the RDF data serialization in this well-known format comes as no surprise [4].

XML uses tags to encode the data attributes, values are then placed between the tag pair. XML allows nesting. RDF/XML is more verbose than other formats and is not considered as human-friendly format for reading. This lead to development of other formats which were more easy to read by the humans. *Listing 14* shows data from *Table 6* expressed using `RDF/XML`.

2.4.6.2 JSON-LD where the LD stands for the *Linked Data* is another format which was defined by W3C [5]. In recent years JSON gained large popularity due for it simplicity and human readability. Although this format did not yet gain widespread use, the ease of use of this format as well as it's ever-growing presence in javaScript API may sway some users toward it. *Listing 15* shows data from *Table 6* expressed using `JSON-LD`.

2.4.6.3 Notation 3 is non-XML serialization format designed specifically with human readability in mind[1][6]. It has set of shorthands for allowing minimizing of commonly used URIs such as `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` using keyword '`a`'. Other features include for example: qualifiers (@forAll), lists, rules, collections and other. *Listing 18* shows off various syntax suggars of the N3. Notation 3 has two major subsets:

2.4.6.3.1 N-Triples is a line-based, plain text format for encoding an RDF graph [10]. Historically it was created before Notation 3 and was ASCII encoded with support for unicode escape sequences, but since then, the specification encourages to use UTF-8 as default encoding. In N-Triples, every information needs to be noted as explicit triple. Because it is most basic format and requires explicit triple, it is used to denote desired production of other serialization formats in W3C documentations. It is subset of N3 but was introduced before it and the N3 format was based around it. *Listing 16* shows data from *Table 6* expressed using `N-Triples`.

The extension of N-Triples called `N-Quads`[11] forms quadruples, where 4th element is used to express the graph to which the triple belongs to.

2.4.6.3.2 Turtle is another subset of `N3` bringing more features to table while still managing to be 'simple enough'[7][8]. It lacks some more complex features of the N3 such as qualifiers and keywords. But allows one to be more expressive then the plain N-Triples notation allowing to save space and help with readability of the document. *Listing 17* shows data from *Table 6* expressed using `Turtle`.

Extension of Turtle called `TriG`[12] has additional notation for GRAPHs, that allows for different contexts to be paced in one document. TRiG is analogous to N-Triples extension N-Quads.

To finish the description of N3 subsets, the following table shows their respectable capabilities:

Feature	Express RDF 1.0	@prefix [], ; a	Collections	Numeric literals	Literal subject	RDF Path	Rules	Formulae
Syntax			(<a> <b>)	2	7 a n:prime.	x!y^z	{?x}=>{?x}	{} @forAll
N3	YES	YES	YES	YES	YES	YES	YES	YES
Turtle	YES	YES	YES					
N-Triples	YES							

Table 1: Comparison of N3 subsets and their features

3 Existing tools

To work with the RDF dataset many tools exists with different capabilities. This chapter describes two of such tools which are close to the implementation described in this thesis. As the thesis implementation language is C, solutions in other languages such as Python, Ruby, javaScript where more tools were found are not present.

3.1 Rapper

Project website	http://librdf.org/raptor/
Initial release	ver. 0.9.0 (2001-01-22)
Current release	ver. 2.0.15 (2014-11-01)
License	LGPL 2.1, GPL 2 or Apache 2

Rapper is part of the Raptor - free software / Open source C library, which contains set of parsers, serializers capable of generating Resource Description Framework (RDF) triples by parsing syntaxes or serialize the triples into a specific syntax. Raptor is designed to be portable and is know to compile across large number of platforms.

As one of its advantages, it allows to directly parse website content, if the *libcurl*, *libxml2* or *libfetch* are present.

Rapper allows to to parse formats specified by RDF 1.1: RDF/XML with latest updates, N-Triples (including legacy support), Turtle (based on 2004 specification), TRiG (with exception of GRAPH and " " omission), RDFa 1.0 & 1.1 (2008, 2012), RSS, GRDDL and microformats.

Its serializer exports to all formats with addition of formats like Atom and JSON.

Project claims to:

- Support all RDF terms including datatyped and XML literals.
- Have optional features including parsers and serialisers can be selected at configure time.
- Have language bindings to Perl, PHP, Python and Ruby when used via Redland.
- Have no memory leaks
- Be fast

3.2 SERD

Project website	https://drobilla.net/software/serd/
Initial release	ver. 0.4.0 (2011-05-24)
Current release	ver. 0.22.3 (2016-03-15)
License	ISC

SERD is C library used to process Turtle and N-Triples files. It doesn't have any external dependencies and compile across compilers and platforms such as Linux, OpenBSD, Mac OSX and Windows.

The SERD is extremely small, as it's size is around 3000 lines of actual code. The *serdi* is standalone tool tool build on SERD library. It is also remarkably fast. In the tests it was fastest tool. The library uses hand-written parser with no intermediate lexer.

Project claims to:

- Have 100% code coverage.
- Have 0 Valgrind memory errors.
- Have no external dependences.
- Have no memory leaks.
- Be fast and lightweight, capable of working in memory constrained environments.

4 The Implementation

The implementation program described in following chapter, is called **ARDP**, which stands for *Another RDF Document Parser*. It is command line utility for Unix-based OSes. ARDP outputs triples to standard output of terminal and errors to the `stderr`. Present version supports the Turtle and N-Triples notations with plans to expand the parser to support TRiG and N-Quads grammars as well.

It is written in the C11 with clang non-standard extensions: blocks and queues. It uses two separate tools t, help with the generation of the parser: *Ragel*, finite-state machine compiler, for the lexer and *Lemon* generator for parser - both are described later in this chapter. The whole project is wrapped as *GNU/Autotools* project which allow easy distribution and deployment of the program on Unix-like platforms.

4.1 Entry considerations

4.1.1 I/O

One of the key capabilities of the implementation was to be the ability to read and process huge files. After several internal test with different approaches using memory mapped files and `fread` with access flags, the conventional method of buffered read proved to be most efficient and portable (as some `fread()` Linux flags were not available on OSX).

Most of the websites providing the data, use compression to minimize the space required to store vast amounts of triples. As such, methods to read compressed files and perform in-memory decompression were explored.

As result, it was decided to abstract reader out of the process, instead the handler is used in lexer, and the user of the API (the implementation) needs to specify the reader.

To help with the reading of compressed files, two libraries were selected: *zLib*, *bzip2*. The first mentioned library allows to read the gzip format archives as well as the plaintext files and as such was chosen to be the default reader in implementation. The second library bzip2 is know for its good compression and ability to recover from some errors; its aligned with the zlib interface, so it was chosen as alternative reader function.

4.1.2 Multi-threading

Most applications nowadays take advantage of the multi-core, multi-thread architecture of the current CPUs. Programming with threads in mind have several consequences such as race condition, restricted access etc.

After exploring POSIX thread library (pthreads), its mechanism didn't sit well with the authors habit of using simpler syntax of blocks in Objective-C. That lead to exploration of option to use blocks in pure C, which turned to be possible with use of clang compiler. Therefore the clang become base compiler of the implementation.

The block runtime is currently not extensively tested on Windows, but with the Visual Studio adoption of clang compiler, blocks ABI implementation is expected. However the implementation was not tested on the Windows platform and additional effort would be required to make it more portable.

4.1.3 Output

The implementation was chosen to be terminal application. It's output is directly printed out to `stdout` and `stderr` streams. To allow maximum output streams, the option `_IONBF`, no buffer, is set before the lexer and parsers start their exucution.

The option was deemed as critical, and the implementation will exit with `EXIT_FAILURE` if the terminal cannot set it.

To improve transparency of the output ANSI escapes and NCurses libraries were explored with the former being integrated into the implementation.

4.2 Used tools

4.2.1 Clang

Clang is compiler front-end for the ever-growing number of languages such as C++, C, Objective-C, CUDA. It's back-end is LLVM[3] project and it's been its integral part since LLVM version 2.6.

The LLVM project started at University of Illinois as a research infrastructure to investigate dynamic compiling techniques. In 2005, Apple Inc. hired Lattner, one of the project directors, to integrate LLVM in Apple development.

Since then Apple switched from GCC to Clang as the main compiler, because it provided better optimisations, clearer error messages etc. More companies such as Google, Sony since joined the clang development and it is currently on par with the GCC compiler in terms of 'raw speed' as well.

This project uses several of clang specific extensions for the C language such is block and queues described later in this chapter.

The code of ARDP was being compiled and tested with optimization profile `-O2`, `pedantic` mode and `-Wall`. The goal was to eliminate maximal number of warnings and errors, but additional flags were required to fine tune some choices made in code (such previously mentioned as use of unsigned char for strings).

[3]Formerly Low-Level Virtual Machine. Currently doesn't mean anything - it's only used as brand.

The optimization profile `-Ofast` was at places more aggressive and removed some of the earlier versions of the wait loops. The code was later optimized to address this issues but was not extensively tested again against this profile. The details of this optimization are described in chapter 4.6.

4.2.1.1 Block

is the clang's version of *closures*. Their are used similarly to normal functions with the key distinction that with addition to executable code they also contain bindings to variables. The example declaration are show in *Listing 2*.

```
int (^)(char, float);
void (^helloWorld)();
```

Listing 2: Block examples

ARDP mainly uses it for function callbacks to allow direct actions on the result of the function, or to modify function behaviour.

4.2.1.2 Grand Central Dispatch

is threading technology developed by Apple Inc. to optimise application support for parallel computing[13]. It is an implementation of task parallelism based on the thread pool pattern.

The programmer creates *queues* on which he runs this code. Simple illustration of the principle is shown in *Listing 3*.

```
dispatch_queue_t queue = dispatch_queue_create( "com.example.unique.id"
    , NULL );

dispatch_async(queue, ^(){
    // This is done on separate thread
    doStuff();
});
```

Listing 3: Grand central Dispatch function

Lexer and Parser in ARDP are running on separate queues and do not share resources appart from API call.

4.2.2 Ragel

To implement the lexer, multiple tools were explored, such as *(f)lex*, commonly used in Unix programs. After experimental explorations, tool called Ragel was chosen for its syntax and ability to include code directly[18].

The Ragel implementation also influenced the choice of the parser, because in contrast to standard approach, implemented lexer is completely independent of the parser. Therefore the lexer calls the parser with the new data. Such approach is called push or event-based parsing. However is not standard practice in parser generators as usually they call the lexer when more data is required (pull parsing).

Ragel can generate several versions of output with different levels of optimisation. The fastest is the `-G2` (goto, in-place actions), which is the format used in the implementation. Ragel is also capable of generation code for binary-search or flat-table, which can help with debugging of the lexer in its design stages.

One of the useful functions of the Ragel is internal support for generation graphViz format graphs of the designed machines, however the Unicode support expanded the generated file and its later image conversion to enormous sizes, and as such needed to be used only on parts of the machine to study particular transformations.

Ragel is the finite-state machine compiler. It supports multiple optimisation levels such table driven vs. goto driven output code for C host language. It allows the code to be executed at arbitrary point in the recognition of the language. The example ragel code is show in *Listing 4*.

```
action dgt      { printf("DGT: %c\n", fc); }
action dec      { printf("DEC: .\n"); }
action exp      { printf("EXP: %c\n", fc); }
action exp_sign { printf("SGN: %c\n", fc); }
action number  { /*NUMBER*/ }

number = (
    [0-9]+ $dgt ( '.' @dec [0-9]+ $dgt )?
               ( [eE] ( [+\-] $exp_sign )? [0-9]+ $exp )?
) %number;

main := ( number '\n' )*;
```

Listing 4: Ragel example code

The requirement of the ARDP to handle UTF-8 string is handled by Ragel by using byte-ranges to define the allowable characters and pushing them along.

The ARDP uses Ragel's capacity to create scanners to work as lexical analyser. Ragel-generated code has no dependencies, and as such can be directly compiled.

In the ragel grammar files few issues needed to be addressed. Fortunately, Ragel provides mechanisms to deal with non-determinalism. The Turtle garmmar required such opetors for strings and iri's as to mark the exit points more clearly.

```
STRING_LITERAL_QUOTE = '"' STR_Q :>> '"' %! { /* Error handing ommited
    */ };
```

Listing 5: Ragel finish guard used in string.

The first example uses the "Finish-Guarded Concatenation" denoted as :>>, which forces the machine one - *STR_Q* - to terminate after machine two - " - is finished.

```
STRING_LITERAL_LONG_QUOTE = '"""' . STR_LQ :> '"""' %! { /* Error
    handing */ };
```

Listing 6: Ragel entry guard used in string.

The second example uses teh "Entry-Guarded Concatenation" denoted as :>, which forces the machine one - *STR_LQ* - to terminate upon entering to machine two - """".

4.2.3 Lemon

As the grammars for `Turtle` and `N-Triples` (including their respective extensions) are both LL(1) and LALR(1), the choice was made to implement the parser using the LALR approach.

LALR(1) stands for Look-Ahead Left-to-Right parser. The number in the braces It is extension of classical LR parser discussed in *Chapter 2.3*, which use the knowledge that at most 1 symbol lookahead is required to successfully judge the rule by parser. In practice it means that the parser is 1 token behind the lexer.

After initial experiments with the hand-written parser, the decision was made to use the parser generator tool to implement the parser, as intricacies of internal workings of parser were hard to implement without full understanding of all underlying concepts.

Because of the way the lexer was beginning to shape, the parser generator needed to be able to perform push-parsing. It also needed to be thread-safe. After exploring the options the *Lemon* parser generator was used as opposed to the more known tools.

Lemon is parser generator created for the purposes of the SQLite project. It consist of two files, which are then compiled into the generator itself, which is then in turn used to generate code from the grammar files. It doesn't have any external dependencies and the project is licensed as public domain, therefore it is ideal for embedding into other project[19].

It generates LALR(1) parsers with output available in C or C++ code. It is similar to another popular parser generator tool Bison (yacc), but has grammar which is less error-prone. The resulting parser is reentrant and thread-safe.

Another difference from Bison is that Lemon is so called push-parser which means that the parser is always called from lexer whereas the Bison calls the lexer itself by default. This also allowed the parser skeleton to be developed prior and separate from development of the lexer, as rules were simply checked against the no value tokens to investigate the parser flow.

Because the ARDP uses lexer on separate asynchronous queue, the Lemon approach is more convenient to implement in this case. The grammar files for Lemon has by convention the same extension as the all YACC clones and similar programmes (*.y).

The Lemon documentation advises to use left recursion in rules for faster reductions and to avoid stack-overflows. It's not supporting common BNF extensions such as ?,* so they need to be implemented by sub-rules.

```
/* EBNF SYNTAX: */
objectList ::=  object (',' object)*

/* Corresponding Lemon rules: */
objectList     ::= object objectList_ast .
objectList_ast ::= objectList_ast COMMA object.
objectList_ast ::= .
```

Listing 7: Lemon rule syntax conventions

The Lemon automatically creates the union for the all the token values by case. However one needs to explicitly state the type within the actions or cast the value. In Lemon documentation, the default type is supposed to be int, but in actual code the default token appear to be (`void*`).

The nonterminals can be associated with the type using `%type` directive, which then automatically casts the nonterminal to it.

```
/* included from custom header file */
   typedef union YYSTYPE {
       float real;
       int  integer;
   } YYSTYPE;

/* actual grammar file */
%token_type {YYSTYPE}
...

/* We have to explicitly identify the type of each token referenced in
 * each action.
 */
coord3D ::= VERTEX COORD(A) COORD(B) COORD(C). {
    coord_3d_t v = {A.real, B.real, C.real};
    ...
}
/* Automatic casting of type associated nonterminals */
%type coord { double }
coord(A) ::= COORD(B). { A = B.real; }

coord3D ::= VERTEX coord(A) coord(B) coord(C). {
    coord_3d_t v = {A, B, C};
    ...
}
```

Listing 8: Lemon terminals and nonterminals type-casting

When Lemon parser receives token, it reduces the tokens currently on the stack as much as possible, and then pushes the new token onto it. To perform actions, Lemon uses symbolic names to reference grammar symbols. Symbolic name needs to start with letter an can be followed by number of letters, numbers or underscores. The symbolic name is replaced globally within the action which can then cause error if the symbols are used eg. in `printf()` function as shown in *Listing 9*.

```
expr(A) ::= expr(B) PLUS expr(C) . { A = B + C; }
text    ::= STRING(s). {
    /* error - %s will be replaced with something like %yymsp[0].minor.
       yy78 */
    fprintf(stderr, "%s", s.value);
}
```

Listing 9: Lemon symbolic name aliases example

Error handling is done in two mechanism. The first mechanism is the syntax error defined by Lemon itself. Upon encountering the syntax error, Parser starts popping out TOKENS until it can again make successful shift. This behaviour can be modified by enabling `YYNOERRORRECOVERY` or `YYERRORSYMBOL`.

The default behaviour, which is currently being used by ARDP, of the error mechanism is to ignore input, until valid input is found. If the `YYERRORSYMBOL` is declared this behaviour changes and the parser starts to pop the tokens until it encounters special nonterminal symbol *error*, shifts it into the stack and the parsing continues. Both strategies warn only about first error and all subsequent messages are ignored until the valid output is produced. The last behaviour, enabled by `YYNOERRORRECOVERY` ignores invalid input but calls the syntax error upon every invalid token. Further more it doesn't call $parser\_failure$ directive nor resets parser.

If the parser encounters the the end of input (token (0,NULL)), *parser_failure* directive is executed. If on the other hand input is shifted out correctly the *parser_success* directive is executed instead. The current implementation of the ARDP shows the parser statistics.

The second mechanism is custom mechanism is embedded into rules, calling the `YYERROR*` macros. This kind error signifies that error occurred while the rule was processed eg. couldn't create sequence, push operation fail and so on. In such cases, the convenience is to try to pass null value and progress with it. As this error is most rare and mostly is caught by next iteration, little data is lost without severe consequences, but as such the more robust technique should also be explored in future work which would use `YYERRORSYMBOL` directive to shift out wrong input.

Lemon also provides tracer function which helps with the debugging of the parser. The ARDP wraps this function into the `ardp_parser_trace()` function, which can be enabled if the application is run with arguments `-enable-parser-debug, -enable-lexer-debug` or argument `-w 4 = -verbose 4`.

4.2.4 GNU Autotools

Autotools is the collective name for the GNU Build system. It is suite of programming tools designated to assist with portability of the code across the Unix systems. It helps with build automation creating standard approach to installing the applications from source files.

Project starts with the specification of file *configure.ac* generated by running command *autoscan* over desired directory. The project is then expanded by adding *Makefile.am* files in subdirectories which describe the behaviour of the *Automake*. In these files the output products are described as well as the options for the compiler for particular file or executable. It also allows to call other tools to transform the sources before the compilation, but integration of the continuous-integration service into project lead to adoption of manual-generation approach.

While writing this thesis, new version of *autotools* came, which announced changes in build output strategy, but ARDP did not yet implement the new structure. As result, several warnings are raised when project is reconfigured.

Autotool's project is usually builded as follows:

```
[cre8iveu@Tomass-MacBook-Pro ~/ardp] > ./configure
[cre8iveu@Tomass-MacBook-Pro ~/ardp] > make
[cre8iveu@Tomass-MacBook-Pro ~/ardp] > sudo make install
```

Listing 10: Installing software using GNU Build system

The script *./configure* prepares necessary files, checks build system for requirements and setup the *config.h* file which contains system specific choices and configurations of the libraries which then are used to enable and disable features in source code while it's being compiled.

The *make* step runs the compiler over all project Makefiles building and linking the project together. After it is finished the application can usually be immediately used from the working directory. To make it available system wide, last step *make install* is run, which on most system also requires elevated privileges as it writes to system paths.

The uninstalling is also easy because one needs to call *make uninstall* to remove the application from system.

Remark 1 When downloading the source from the GIT repository, additional step is needed to prepare the project:

```
[cre8iveu@Tomass-MacBook-Pro ~/ardp] > ./autogen.sh
```

which is script adding dependencies and preprocessed files which are ignored by the git.

4.2.5 Travis-CI

Continuous integration is the method proposed by Grady Booch in 1991. It was adopted as a part of extreme programming, which advocates integrating more than once a day. The main goal is to detect regression of codebase as soon as possible, using automated tests.

Travis-CI is one of the services for the continuous integration. It's free for open-source project hosted on Github. Many of leading open-source project such as FFMpeg, VLC and others take advantage of it by testing each commit and pull request against multiple configurations and operation systems. Travis-CI works with the virtual containers which run various versions of OS. At the time of the writing of this thesis Travis-CI supports OSX and Ubuntu as their free platforms.

ARDP take advantage of Travis-CI, in testing it's codebase against both available platforms. The ARDP was developed on OSX machine so compatibility issues were sorted using Travis-CI without the need to switch into the linux prior the the more extensive tests.

4.3 Helper structures

Before the main structures of the program will be discussed, several helper structures and functions will be discussed. Particularly the string handling, hashmap and sequence.

In order to allow the better readability of the triples, the auxiliary functions were created to allow the formatting of the string as well as 'fat-pointer' implementation of library to handle the stings. All following references to header files are to project header files found in *include/ardp*

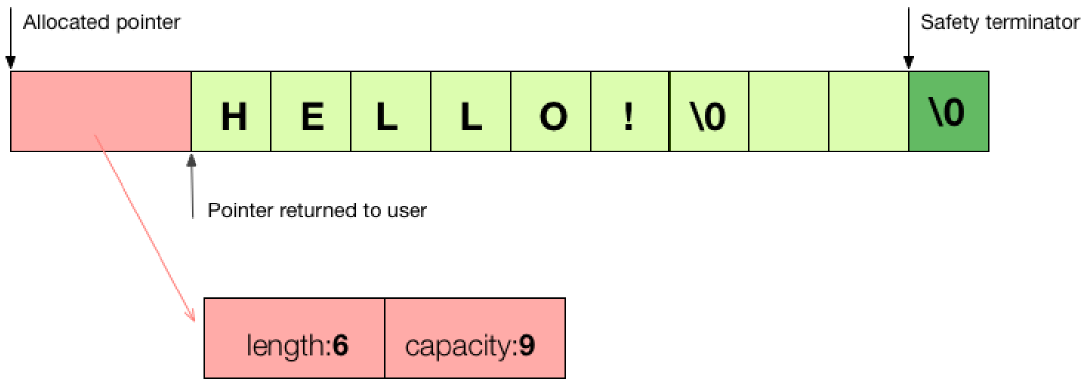

Figure 4: Illustration of principle behind the fat-pointer string library.

folder. The commonly used programming structures used in project, namely: `hashmap` and `sequence` are well described in programming textbooks, and can be found in project's implementation headers *hashmap.h* and *sequence.h* respectively.

4.3.1 String handling

The pure C doesn't have proper string handling library in it's core libraries. In order to simplify the manipulation with strings, library *string.h* was created. It is safer then pure string buffer handling as string can handle the string buffer with the \0 terminators inside.

The ARDP's string library uses memory management trick called 'fat-pointer' to handle string and its operation. Library was inspired by the Violla library which uses the 'far-pointers' to simulate objects in the pure C language.

The *string* consists of the header, in which the extra informations about the string, specifically the string's capacity and length, and the string buffer.
Because the ARDP 'fat-pointer' string implementation is opaque to programmer, to enhance clarity of the source code, string's plain type `uint8_t*` is redefined as `utf8`. It is then used to distinct places in code, where the 'fat-pointer' string is required.

The library allow the string manipulation by two types of manipulations: by characters and by strings. Several functions are dedicated to appending and prepending to string.

4.3.2 Colored output

To help with the visual perception of the output to terminal, the *color.h* header file specifies the wrappers for `stdio` function `fprintf`.

Color in terminal is expressed by using **ANSI escape codes**. As not all terminals are capable of displaying such escapes correctly, or the 'piping' is used, the ARDP color library check capability of terminal from environment `TERM` variable (Its value represents capability of particular terminal or its emulator; possible values are xterm, dummy, term, vt100 etc) and sets the global variable `color_stdout_is_tty` (variable is defined as external and therefore needs to be present in the main of the source code, this allows the direct manipulation of the library behaviour.

ANSI escape codes begin with ASCII *ESC* character (0x1B hex) followed by escape specifier. To use the colors the **Control Sequence Indicator** '[' is used, particularly its **Select Graphic Rendition** parameters. ARDP uses 1 (bold) and 0 (reset) as well two ranges of SGR codes:

- 30-37 for foreground color
- 40-47 for background color

Example 6
To color print text "The quick brown fox jumps over the lazy dog." the actual C string with ANSI escapes will be `"\x1B[31mThe quick brown fox jumps over the lazy dog.\033[0m"`.

■

4.4 RDF data structures

Parser uses internally structures `rdf_term`, `rdf_statement` and `sequence`. The input token type is set to be `utf8` which is in fact typedef for the `uint8_t*` to note use of the string creating using ARDP's string library.

4.4.1 Term

To express the RDF term, `struct rdf_term` is used. It is defined in *rdf.h* header file. It wraps the union with type: `URI`, `CURIE`, `BLANK`, `LITERAL`.

Internally, `URI` and `CURIE` are identical - the notation only distinguishes the ability to expand the URI from the prefix namespaces as opposed to base address.

`LITERAL` is structure which holds the string itself as *langtag* or *datatype*, both of which are optional, but only one or none can be used at the same time.

4.4.2 Statement

Finished triple is created in parser using the `struct rdf_statement`. It is defined in *rdf.h* header file. It was designed with the expansion for TRiG and N-Quads syntax in mind and as such holds 4 terms: `subject`, `predicate`, `object`, `graph`.

Current parser is only capable of parsing the triples, so the graph is always set to NULL.

4.5 Program structure

ARDP lexer and parser are independent and both are managed via the main application. The implementation to simplify the code, implements both of them as global structures opaque to the programmer. *Listing 11* demonstrates basic setup of the ARDP runloop without preparation stage or threading code.

```
/* GETOPT code */
...
ardp_parser_create();

ardp_lexer_create();
ardp_lexer_defaults();
{
    struct ardp_lexer_config cfg;

    cfg.logging.level = DEBUG;
    cfg.logging.eprintf = &lexer_error;

    cfg.cb.stoken = ^int( int type, const char *_Nullable value,
        size_t line, size_t col){
            // Push token to parser
            ardp_parser_exec(type, value, line, col);
            return ARDP_SUCCESS;
    };
    ardp_lexer_init(&cfg);
}
...
ardp_lexer_reader( reader_callback , file );
...
/* CLEANUP code */
```

Listing 11: ARDP lexer - parser interaction

ARDP will fire token callback block asynchronously, as soon as possible, on separate serial queue managed internally by the lexer. The programmer is then provided with ability to change or modify the incoming data and to pass them to parser by calling `ardp_parser_exec()` function.

Because the ARDP lexer and parser are called in different then main thread, application terminated as soon as it entered the lexer. To prevent this the guard-loop was put in place and was later optimised[4] and in the end solved by using `dispatch_group`.

4.5.1 Lexer

Lexer in ARDP is created from by preprocessing the Ragel file. For distribution purposes this file is already created in ARDP package. To regenerate it, one need to run `ragel turtle.rl -G2 turtle.c` from inside the `src/lib/lexers` directory.

Lexer contains two serial queues: lexer & event. While lexer is being run on the the lexer queue. The tokens and errors are being send to further processing on event queue, allowing parsing to continue uninterrupted while other parts of the application works with the token.

The current implementation exits lexing functions as soon as possible, to modify this behaviour `dispatch_group` should be used. The current implementation guard against premature exit by waiting for the parser queue to finish.

4.5.2 Parser

The parser is preprocesses for distribution, the same way as the lexer. To regenerate it, one need to run `lemon turtle.y` from inside the `src/lib/parsers` directory.

One of the side-effects of using lemon is that that it generates its own token type numbering, which was then translated back into hand-written definition created for lexer. Expansion of the parser capabilities to include terminals for graphs would require manual rewrite of token type definitions.

4.6 Optimization

Altho the ARDP uses multiple queues to perform various operations asynchronously, the user interface is synchronous. In order to achieve this, the wait loop were implemented at critical places. While profiling the application, those loops, which were essentially no-op, were thrown away with the *-O3* compiler optimization profile, and were using CPU with lower optimization profiles. To address this issue, the loop was rewritten to use unix *unistd.h* function `usleep()`.

The ARDP functions requiring the wait for the completion of the underlying operations, such as `ardp_lexer_process_reader()` are then written as while loops with sleep. The `usleep` function allows the CPU thread to be idle for the specified duration. The while loop then rechecks the conditions.

```
while ( dispatch_queue_isempty(queue) ) {
    usleep(20);
}
```

[4] See *Chapter 4.6.2*

Listing 12: ARDP idle loop example

The exact wait time was determined experimentally. By further optimising the code, the need for explicit wait-loop was eliminated using `dispatch_group`.

4.6.1 Memory management

The ARDP application uses fully manual memory management. As the most of the operations requiring memory are string operations, the global policy to use `calloc()` when possible, was adopted.

The `malloc()` function is not guaranteed to to be initialized to zeros. In such cases the use of another function `memset()` is required, which forces the the virtual memory system to map the corresponding pages into physical memory in order to zero-initialize them. The `calloc()` function on the other hand reserves the required virtual address space, but defers the initialization until the memory is actually used.

The early version of the ARDP also used the explicit `malloc_zone_malloc`, but as the function was not implemented in linux, the approach was later abandoned.

As the ARDP implements own string implementation, the `memcpy()` function is used in favour of `strcpy()` even if the string library would allow for its use. By using `memcpy()`, clang compiler can decide to inline the function to use the instruction to directly copy the data by value[21].

4.6.2 Clang optimisations

As ARDP implements the lexer and parser as shared opaque structures hidden to programmer, the functions often checks, if the corresponding structure is initialized. But because one expects that if the function is being called, all previous errors were resolved, the linux kernel macros are used to advise the compiler in branching.

If the result of the branching is as predicted, it leeds to slight speed improvement. The downside of this approach is, that if the prediction is not met, the call suffers additional penalty, as the branch needs to be reentered and computed from scratch.

The branching prediction is achieved using the compiler macro `__buildin_expect` as shown in Figure 13.

```
#define likely(x) __builtin_expect(!!(x), 1 )
#define unlikely(x) __builtin_expect(!!(x), 0 )
```

Listing 13: Linux kernel branch prediction macros

The clang compiler also has compiler feature called nullability, it advises the compiler to check the pointers in static analysis and helps to resolve some pointer related errors. Clang defines three nullability states: `_Nullable`, `_Nonnull`, `_Null_unspecified`.

Depending on the function type and purpose, one of the qualifiers should mark the function argument. As this feature was introduced in later clang versions and TRAVIS-CI sometimes had problem with this feature on linux VM, clang's compiler macro `__has_feature(nullability)` has been used with defines for platforms without this feature.

The another optimization provided by the clang compiler is vectorization. The compiler has special flags to analyze the loops to determine if they can be vectorized called `-Rpass-analysis= loop-vectorize`.

In the project vectorization is turned on by default, but the further work would be required in optimizing the loops, as most of them were marked by compiler as variable with no determined length. This option also helps to unroll loops if they can be replaced by series of the if-else statements to improve performance.

The function inlining, analyzed using clang flag `-Rpass-analysis= inline` marked many of the functions as candidates for inlining, but the choice was left to the compiler as code with marked inlines wasn't yielding different results with optimization profile enabled.

4.6.3 Intel's SIMD Instruction set

The another approach to speed the application was to explore Intel's SSE instructions. Streaming SIMD Extensions is extension to the X86 instruction set, performing Single-Instruction Multiple-Data operations [20]. It was designed by Intel for their processor Pentium III in 1999 as an answer to AMD's 3DNow! instruction set. It came with 70 new instructions aimed primarily at single precision floating point data. The aim was to speedup digital signal and graphical processing, where one operation is performed multiple times on different sets of data.

The SSE instructions were improvement upon previous MMX SIMD instruction set introduced for processor Pentium in 1997. The MMX instruction extension defined 8 registers denoted MM0 through MM7 and set of operations to perform on them. Each register is 64 bits wide. The limitation was that all the operation were integer operations. Another limitation was that to avoid problems with OS at the time, the MMX registers were only aliases for FPU registers, thus blocking CPU from performing floating-point computations. The difference to FPU x87 registers was, that the MMX registers were directly addressable.

With the introduction of the SSE, Intel created 8 new 128-bit wide registers called XMM0 through XMM7 which were now independent of the FPU registers. The focus of the SSE was on single-precision floating-point operations. The separation allowed applications to mix SIMD and floating-point operations in single application without expensive switch.

The next version, SSE2, which came with the PIII successor Pentium IV, extended instruction available to be performed on XMM registers allowing for operations on integers (8,16,32 bit) as well as double precision floating-point operations as opposed to only single-precision floating-point (32 bit) operations available in SSE.

The newest iteration of the SSE, the SSE4, is the first instruction extension set which contains operations which are not specific to multimedia applications and DSP - specially with the focus on string manipulation introduced in SSE4.2 - STTNI (String and Text New Instructions).

In the case of the ARDP, the string manipulations, particularly substring searching, were marked as candidates for the SSE optimization.

The SSE instructions can be included in application using two methods: assembly (with or without preprocessor) or in form of intrinsics. As the author, doesn't have much experience with programing in x86 assembler, the intrinsic approach was taken.

By testing were various implementations of the `strchr()` and `strstr()` functions using intrinsics with different results. The final solution uses older SSE2 set of instructions as opposed to newer SSE4.2 instruction set, offering string manipulation instructions, because of the simplicity of SSE2 version as opposed to the SSE4.2 version where the corner cases needed to be checked, adding instructions. The SSE4.2 were outperforming the SSE2 on longer string, but the majority of tested URIs were rather short and the performance gain was not significant. With more research into the the SSE the further optimisation is highly likely.

5 Evaluation

This chapter describes the state of the final implementation, its limitations and performance. It also offers some suggestions into the way to improve the current solution in retrospective.

5.1 Format conformance

The parser aligns with the Turtle 1.1 specification. It is also capable of parsing the N-Triples format. The implementation is also ready to expand the parser to TRiG parser if the graph nonterminals are added.

One of the areas where the ARDP need the improvement to increase the format conformance is the lexer. The current error handling is quite primitive, which leads to sometimes wrong error recovery in parser. The lexer detects the error and if the option is specified, displays the error, but the recovery strategy is not properly explored.

One of the ability of the Ragel is to specify multiple machine entry points which should be implemented late to invoke the strict N-Triples mode. The current implementations treats N-Triples documents as full Turtle documents, which should throw error as stated per N-Triples specification.

The current parser marks the error line and the token which is deemed as error. The column detection is not yet implemented as the column pointer implementation has boundaries issues when new block of data was read. The current lexer then post column as 0.

One of added features of the ARDP is ability do directly read from compressed files by doing in-memory decompression. The implementation provides two variants of read function: zlib's and bzip2's versions. The introductory tests into integrating the CURL library for direct-from-web reading were conducted, but are not included in this thesis implementation release.

5.2 Limitations

The output of the ARDP is printed directly to the terminal. With the +10Gb test file, the terminal, trying to keep hold of the history hung-up. One of the solutions was to pipe the output to the file and then explore the results. Also the tools to measure performance were significantly impacted upon processing such large data.

In order to see the results and perform the test multiple times, the files smaller then 1GB were taken in test. As larger files tend to take considerable time to be parsed and multiple run measure records were taking large amount of space.

5.3 Performance

In the performance comparisons the ARDP was stuck agains the two tools mentioned in previous chapter: SERD an RAPPER.

5.3.1 Testing datasets datasets

To test the performance of the tools, following 4 datasets were chosen:

ID	Resource name	Filename	Size (MB)	Sampled
DS1	National Digital Library	`a0005.ttl`[5]	96	2016-02-16
DS2	Rijksmuseum collection	`201509-rma-edm-collection.ttl`[6]	727.5	2015-09-16
DS3	YAGO Scheme	`yagoScheme.ttl`[7]	0.04	2013-01-29
DS4	USPTO Patent data sample	`2013/ipg130101.nt`[8]	188	2013-05-11

Table 2: Datasets used in tests

The *DS1* needed to be fixed in order for the tools to process it, as it contained error escapes in the some URIs. The *DS3* proven to be quite small and on the edge of the clock granularity, resulting in test be run multiple times as one test suite, with the time divided then to get unit timing.

5.3.2 Testing environment

All of the programs for parsing the RDF were run with default or recommended settings on Macbook Pro Late 2013 equipped with Quad-core Intel Core i7 CPU, model 4750HQ, with 22nm Haswell/Crystallwell architecture; 8Gb DDR3L SDRAM memory, SSD PCIe hard drive.

Used and tested operating system included Mac OSX 10.10 Yosemite & 10.11 El Captain up to 10.11.5 Developers beta on which the performance was measured. The other tested system was Ubuntu 15.10 and Ubuntu 16.04 (Daily build). All operating system were run using 64 bit variants.

On the OSX the installation of XCode and its command line tools (CLT) installs all necessary libraries and dependencies. On Ubuntu machine, additional installation of zlib, bzip and libdispatch was necessary.

5.3.3 Testing methodology

To measure the application performance the CPU timer based used in XCode's profiling API was used to obtain more accurate times. The timer reported *wall-clock* time of each process. The

[5]source: http://data.theeuropeanlibrary.org/download/opendata/a0005.ttl.gz

[6]source: http://rijksmuseum.sealinc.eculture.labs.vu.nl/rdf/201509-rma-edm-collection.ttl.gz

[7]source: http://resources.mpi-inf.mpg.de/yago-naga/yago/download/yago/yagoSchema.ttl.7z

[8]source: http://us.patents.aksw.org/export/2013.zip

Application	command with parameters
ARDP	`ardp -c never` *filename*
SERDI	`serdi` *filename*
RAPPER	`rapper -i turtle -o ntriples` *filename*

Table 3: Setting used in the test

Application	DS1	DS2	DS3	DS4
ARDP	19.12	254.32	0.02	26.43
SERDI	7.65	88.21	0.01	11.68
RAPPER	18.01	208.36	0.02	27.46

Table 4: Average wall-clock runtime in seconds

granularity of the timer was 10ms. All the output was piped to `/dev/null` to avoid buffering delays by the terminal itself.

Each application was run 10x over particular dataset and the average execution time was calculated. Memory statistics were provided automatically by the *Instruments* tool.

5.3.4 Results

The results are presented in *Table 3*, *Table 4* and *Table 5*.

5.3.5 Result discussion

In result ARDP made good use of memory, but threads operations were not optimised properly, yet. As result, ARDP resulted in number of forced context switches and was more CPU expensive.

The SERDI with hand-written parser was extremely fast. It's code is making shallow copies of objects passing only pointers, making copies only when completely necessary. Its optimised performance was cooping well with all of the files, but as it is full-featured parser-serializer, it was not tolerant to URI errors and quit prematurely. In some of this cases `-l` allowed more lazy rules, but was not always enough.

The RAPPER has the most features and runs faster then the ARDP, but consumes large amounts of memory. Furthermore it had problems with some files which were larger then 4GB. It's error recovery mechanism was little more forgiving than in SERDI, but still could prevent file from being parsed especially on URI errors such as spaces.

Application	DS1	DS2	DS3	DS4
ARDP	1.3	1.3	0.9	1.1
SERDI	0.5	0.4	0.3	0.4
RAPPER	186	1360	10	372

Table 5: RAM Memory usage in MB

6 Conclusion

The aim of this thesis was to create parser for the RDF data, which would be able to take big files as the input, support UTF-8 and be capable of basic error signalisation and recovery. As such, the implementation fulfilled those requirements.

Additional features such as reading from compressed files and doing in-memory decompression, add value to it, but it still has huge room to improve its performance. Particularly the multithread code should be in-depth profiled and optimised to prevent so much context-switching as its burning away its performance.

After the profiling of the ARDP application, the most performance critical are the deep copies of the objects and context switching for the threaded code which would be the objectives for improvement in future work.

In order for current implementation to preserve the data, one token is deep-copied multiple times until its transformed into the acceptable structure without preprocessing it on the way. This performance penalty as each new memory allocation requires system call to obtain it while then old memory is simply being freed after coping the data into new structure. More elegant solution is to make shallow copies or introduce reference counting.

Other areas which could be explored with future work:

- Add other formats such as JSON and XML RDF notations.
- Expand the current parser to support full N3 and TRiG, N-Quads grammar.
- The current error handling and reporting is somehow limited. Thus author would recommend to expand it to have more context like clang warning with suggestions.

In order to expand the current implementation to support the TriG grammar, the lexer needs to be expanded by one additional keyword `GRAPH`. The parser needs 7 new rules in addition to the current Turtle rules to deal with the graph sugar syntax. The revised grammar of the TRiG aligns itself more with the Turtle, allow seamless integration into the parser.

Similarly the grammar for the N-Quads, requiring 2 new rules, would be easily included into the parser. The N3 notation was not yet fully standardised at the time of the writing of this thesis, therefore no estimation is given as to its integration.

The main problem with the integration of the new grammars is the need for the format separation. This could be achieved at the lexer level by introducing multiple entry points into the Ragel machine. In parser, the expansion would require change of the error-handling mechanism from its default mechanism into the custom one.

This error-handling mechanism would then better react on errors in syntax, as current mechanism ignores the errors inside the rules (they are handled using different mechanism).

References

[1] *Notation3.* In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 20012 [cited 2016-04-22]. Available from: `https://en.wikipedia.org/wiki/Notation3`

[2] *RDF1.1 Primer.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cit. 2016-02-13]. Available from: `https://www.w3.org/TR/rdf11-primer/`

[3] *RDF1.1 Concepts and Abstract Syntax.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-02-15]. Available from: `https://www.w3.org/TR/rdf11-concepts/`

[4] *RDF 1.1 XML Syntax.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-03-22]. Available from: `https://www.w3.org/TR/rdf-syntax-grammar/`

[5] *JSON-LD 1.0.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-03-22]. Available from: `https://www.w3.org/TR/json-ld/`

[6] *Notation3 (N3): A readable RDF syntax.* In: World Wide Web Consortium [online]. (USA): W3C, 2011 [cited 2016-04-20]. Available from: `https://www.w3.org/TeamSubmission/n3/`

[7] *Turtle - Terse RDF Triple Language.* In: World Wide Web Consortium [online]. (USA): W3C, 2011 [cited 2016-03-25]. Available from: `http://www.w3.org/TeamSubmission/turtle/`

[8] *RDF 1.1 Turtle: Terse RDF Triple Language.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-03-25]. Available from: `https://www.w3.org/TR/turtle/`

[9] *RDF1.1 Errata.* In: World Wide Web Consortium [online]. (USA): W3C, 2016 [cited 2016-03-23]. Available from: `https://www.w3.org/2001/sw/wiki/RDF1.1_Errata`

[10] *RDF 1.1 N-Triples: A line-based syntax for an RDF graph.* In: World Wide Web Consortium [online]. W3C, 2014 [cited 2016-04-25].
Available from: `https://www.w3.org/TR/2014/REC-n-triples-20140225/`

[11] *RDF1.1 N-Quads.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-02-15]. Available from: `https://www.w3.org/TR/n-quads/`

[12] *RDF 1.1 TriG: RDF Dataset Language.* In: World Wide Web Consortium [online]. (USA): W3C, 2014 [cited 2016-02-15]. Available from: `https://www.w3.org/TR/trig/`

[13] *Grand Central Dispatch (GCD) Reference.* Mac Developer Library [online]. USA: Apple, inc., 2014 [cited 2016-03-02]. Available from: `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`

[14] *What is Parser?,* Techopedia. [online]. 1.3.2016 [cited 2016-03-12].
Available from: `https://www.techopedia.com/definition/3854/parser`

[15] JIANG, Tao, Ming LI, Bala RAVIKUMAR and Keneth W. REGAN. *Formal Grammars and Languages* [online]. New York, 1999 [cited 2016-03-25].
Available from: `http://www.cs.ucr.edu/~jiang/cs215/tao-new.pdf`

[16] AHO, Alfred V. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Boston: Pearson-Addison Wesley, 2007. ISBN 03-214-8681-1.

[17] AHO, Alfred V. and Jeffrey D. ULLMAN. *The theory of parsing, translation, and compiling: Volume I: Parsing.* 1st ed. Englewood Cliffs, N.J.: Prentice Hall, 1972. ISBN 01-391-4564-8.

[18] *Ragel State Machine Compiler: User Guide* [online]. Adrian Thurston [cited 2016-04-25].
Available from: `https://www.colm.net/files/ragel/ragel-guide-6.9.pdf`

[19] *The Lemon Parser Generator.* Hwaci software [online]. (USA): Hwaci, 2001 [cited 2016-04-25]. Available from: `http://www.hwaci.com/sw/lemon/lemon.html`

[20] *Intel Multimedia Instructions: (MMX, SSE, SSE2, S2, SSE3, SSSE3 and SSE4).* UCONN [online]. Storrs, Connecticut: UCONN, 2011 [cited 2016-04-25].
Available from: `http://www.engr.uconn.edu/~zshi/course/cse5302/student/mmx.pdf`

[21] *Tips for Allocating Memory.* Mac Developer Library: Memory Usage Performance Guidelines [online]. (USA): Apple, inc., 2013 [cited 2016-04-25]. Available from: `https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html`

A RDF formats listings

Subject	**Predicament**	**Object**
ex:Student	ex:studiesAt	ex:schools#VSB
ex:Student	ex:login	"xyz123"
ex:Professor	ex:teaches	ex:Student

Table 6: Example RDF data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.com/#">
  <rdf:Description rdf:about="http://example.com/Student">
      <ex:studiesAt>http://example.com/schools#VSB</ex:studiesAt>
      <ex:login>xyz123</ex:login>
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/Profesor">
      <ex:teaches>http://example.com/Student</ex:teaches>
  </rdf:Description>
</rdf:RDF>
```

Listing 14: Serialization of example data to RDF/XML

```json
{
      "http://example.org/Student" : {
            "http://example.org/#studiesAt" : [{
                "value": "http://example.com/schools#VSB", "type": "uri"
            }],
            "http://example.org/#login" : [{
                "value": "xyz123", "type": "literal"
            }]
      },
      "http://example.org/Profesor" : {
            "http://example.org/#teaches" : [{
                "value": "http://example.com/Student", "type": "uri"
            }]
      }
}
```

Listing 15: Serialization of example data to JSON-LD

```
<//example.com/Student> <//example.com/#studiesAt> <//example.com/
    schools#VSB> .
<//example.com/Student> <//example.com/#login> "xyz123" .
<//example.com/Profesor> <//example.com/#teaches> <//example.com/
    Student> .
```

Listing 16: Serialization of example data to N-Triples

```
@prefix ns0: <http://example.com/#> .

<http://example.com/Student>
  ns0:studiesAt "http://example.com/schools#VSB" ;
  ns0:login "xyz123" .

<http://example.com/Profesor> ns0:teaches "http://example.com/Student"
    .
```

Listing 17: Serialization of example data to Turtle

```
@prefix : <foo.n3#>. # Local stuff
@forAll :d, :x, :y, :F, :G.

{ <localFile.n3> log:semantics :F .
  :F log:includes { :theSky :is :blue}
}

log:implies { :test10a a :success } .
```

Listing 18: Example of data serialized in N3 format

B Content of the CD

The CD directory tree looks as follow:

```
$root
  thesis
    source
      thesis.tex
      diploma.cls
      Figures
      SourceCodes
    thesis.pdf
  implementation
    ardp-*.tar.gz
    ardp
      INSTALL
      README
      src
      include
      ...
  support
```

The **thesis** folder contains source the LaTeX source code of this thesis, as well as its digital copy.

The **implentation** folder contains the archive made be running autotools *make distro* suitable for normal installation from archive. The **ardp** folder contains the mirror of the git repository (tag marked as 'thesis'). The **INSTALL** file contains the steps and prerequisites necessary to build the ARDP project.

The **support** folder contains various supplementary files such as source code for the tools used (ragel, lemon, libdispatch).